SuRF Documentation

Release 1.1.4

Cosmin Basca

April 12, 2011

CONTENTS

SuRF is an Object - RDF Mapper based on the popular *rdflib* python library. It exposes the RDF triple sets as sets of resources and seamlessly integrates them into the Object Oriented paradigm of python in a similar manner as *ActiveRDF* does for ruby.

Quick start:

```
from surf import *

store = Store(  reader='rdflib',
            writer='rdflib',
            rdflib_store = 'IOMemory')

session = Session(store)

print 'Load RDF data'
store.load_triples(source='http://www.w3.org/People/Berners-Lee/card.rdf')

Person = session.get_class(ns.FOAF['Person'])

all_persons = Person.all()

print 'Found %d persons that Tim Berners-Lee knows'%(len(all_persons))
for person in all_persons:
    print person.foaf_name.first

#create a person object
somebody = Person()
somebody_else = Person()

somebody.foaf_knows = somebody_else
```

CHAPTER

ONE

DOCUMENTATION

1.1 Installing SuRF

SuRF can be most easily installed with setuptools (setuptools installation), by running this from command-line:

```
$ easy_install surf
```

alternativelly, *SuRF* can be downloaded and the following command executed:

```
$ sudo python setup.py install
```

if you choose the second option, than the dependencies must be installed beforehand. *SuRF* depends on *rdflib* and *simplejson*.

1.1.1 Installing rdflib

SuRF depends on *rdflib* version 2.4.x or 3.x.x. On **Windows** platforms the *rdflib* package requires python to be configured with a c/c++ compiler in order to build native extensions. Here are the steps to required set up *rdflib* on Windows:

1. Download and install *MinGW* from http://www.mingw.org/
2. Make sure *gcc* is installed
3. Add the *[MinGW]\bin* folder to system Path
4. Edit (create if it does not exist) the following file *[Python 2.X dir]\lib\distutils\distutils.cfg*:

   ```
   [build]
   compiler = mingw32
   ```

5. Run this from command-line (or simply install *surf* - it will install *rdflib* for you automatically):

   ```
   $ easy_install rdflib>=2.4.2
   ```

Further information can be found here:

- http://code.google.com/p/rdflib/wiki/SetupOnWindows

1.1.2 Installing SuRF plugins

SuRF can access and manipulate RDF data in several different ways. Each data access method is available in a separate plugin. You can install all or just some of the plugins. Currently available plugins are:

- *The sparql_protocol Plugin*. Use this plugin to access data from **SPARQL HTTP** endpoints. Install it by running this from command-line:

```
$ easy_install -U surf.sparql_protocol
```

- *The rdflib Plugin*. This plugin uses *rdflib* for data access and manipulation. Install it by running this from command-line:

```
$ easy_install -U surf.rdflib
```

- *The allegro_franz Plugin*. Use this plugin to access **Franz AllegroGraph** triple store. Install it by running this from command-line:

```
$ easy_install -U surf.allegro_franz
```

- *The sesame2 Plugin*. Use this plugin to access data using **Sesame2 HTTP** protocol. Install it by running this from command-line:

```
$ easy_install -U surf.sesame2
```

1.1.3 Loading plugins from path or running *SuRF* in embedded mode

In the cases where *SuRF* is distributed bundled with an application, one can choose to load the plugins from a specific location. You can do so via the `surf.plugin.manager.add_plugin_path()` method, as in the code snippet below:

Note: In order to run the following code snippet, one needs to generate the **egg-info** directory if not present, this can be done with the following command:

```
$ python setup.py egg_info
```

```
from surf.plugin import manager

#setup a local folder where the plugins are stored
manager.add_plugin_path('/path/to/plugins')
# reload plugins if, allready loaded
manager.load_plugins(reload=True)

# the rest of the application logic
...
```

1.1.4 Setting up *SuRF* in development mode

To get the latest development version of *SuRF*, check it out from subversion and install it using the *setup.py* script. Plugins live in the same subversion tree but each has it's separate *setup.py* script, so they need to be installed separately.

Instructions for getting the code from subversion can be found here:

http://code.google.com/p/surfrdf/source/checkout

Here is a brief and useful list of **commands** for building eggs, installing in development mode and generating documentation:

Command	Task
python setup.py bdist_egg	Build the SuRF egg file
python setup.py bdist_egg register upload	Build and register with *pypi* SuRF if you have access rights
python setup.py develop	Install SuRF in development mode
make.bat html	regenerate the documentation

1.2 Quick Start examples

1.2.1 Using the public SPARQL-endpoint from DBpedia

Getting Phil Collins albums and covers:

```
import surf

store = surf.Store(reader = 'sparql_protocol',
                   endpoint = 'http://dbpedia.org/sparql',
                   default_graph = 'http://dbpedia.org')

print 'Create the session'
session = surf.Session(store, {})
session.enable_logging = False

PhilCollinsAlbums = session.get_class(surf.ns.YAGO['PhilCollinsAlbums'])

all_albums = PhilCollinsAlbums.all()

print 'Phil Collins has %d albums on dbpedia' % len(all_albums)

first_album = all_albums.first()
first_album.load()

print 'All covers'
for a in all_albums:
    if a.dbpedia_name:
        cvr = a.dbpedia_cover
        print '\tCover %s for "%s"' % (str(a.dbpedia_cover), str(a.dbpedia_name))
```

1.2.2 Loading a public remote RDF file using rdflib

Print all persons mentioned in Tim Berners-Lee's FOAF document:

```
import surf

store = surf.Store(reader = "rdflib",
                   writer = "rdflib",
                   rdflib_store = "IOMemory")

session = surf.Session(store)

print "Load RDF data"
store.load_triples(source = "http://www.w3.org/People/Berners-Lee/card.rdf")

Person = session.get_class(surf.ns.FOAF["Person"])
```

```
all_persons = Person.all()

print "Found %d persons in Tim Berners-Lee's FOAF document" % (len(all_persons))
for person in all_persons:
    print person.foaf_name.first
```

1.2.3 Connecting `surf.store.Store` to MySQL

This code was contributed by Toms Baugis

```
import rdflib
import surf

# mysql connection string that will be passed to rdflib's mysql plugin
DB_CONN = 'host=localhost,user=surf,password=password,db=rdfstore'


def get_rdflib_store():
    store = rdflib.plugin.get('MySQL', rdflib.store.Store)('rdfstore')

    # rdflib can create necessary structures if the store is empty
    rt = store.open(DB_CONN, create=False)
    if rt == rdflib.store.VALID_STORE:
        pass
    elif rt == rdflib.store.NO_STORE:
        store.open(DB_CONN, create=True)
    elif rt == rdflib.store.CORRUPTED_STORE:
        store.destroy(DB_CONN)
        store.open(DB_CONN, create=True)

    return store

store = surf.Store(reader='rdflib',
                   writer='rdflib',
                   rdflib_store = get_rdflib_store())
session = surf.Session(store)
```

1.2.4 Connecting `surf.store.Store` to AllegroGraph

```
import surf
import threading

store = surf.Store(reader = 'allegro_franz',
                   writer = 'allegro_franz',
                   server = 'localhost',
                   port = 6789,
                   catalog = 'repositories',
                   repository = 'surf_test')

print 'Clear the store if supported'
store.clear()

print 'Create the session'
session = surf.Session(store,{})
#session.enable_logging = True
```

```
#session.use_cached = True

print 'Define a namespace'
surf.ns.register(surf='http://surf.test/ns#')

print 'Create some classes'
Actor = session.get_class(surf.ns.SURF['Actor'])
Movie = session.get_class(surf.ns.SURF['Movie'])

print Actor, Actor.uri
print Movie, Movie.uri

print 'Create some instances'
m1 = Movie('http://baseuri/m1')
m1.surf_title = "Movie 1"

m2 = Movie('http://baseuri/m2')
m2.surf_title = "Movie 2"

m3 = Movie('http://baseuri/m3')
m3.surf_title = "Movie 3"

m4 = Movie('http://baseuri/m4')
m4.surf_title = "Movie 4"

m5 = Movie('http://baseuri/m5')
m5.surf_title = "Movie 5"

a1 = Actor('http://baseuri/a1')
a1.surf_name = "Actor 1"
a1.surf_adress = "Some drive 35"
a1.surf_movies = [m1, m2, m3]

a2 = Actor('http://baseuri/a2')
a2.surf_name = "Actor 2"
a2.surf_adress = "A different adress"
a2.surf_movies = [m3, m4, m5]

# saving
print 'Comitting ... '
session.commit()
print 'Size of store ', session.default_store.size()

print 'Retrieving from store'
actors = list(Actor.all())
movies = list(Movie.all())

print 'Actors : ', len(actors)
print 'Movies : ', len(movies)

print 'Actor 1 cmp: ', a1 == actors[0]
print 'Actor 1 cmp: ', a1 == actors[1]
print 'Actor in list : ', a1 in actors

print 'All movies %d' % len(movies)
for m in movies:
    print m.surf_title
```

```
print 'All actors %d' % len(actors)
for a in actors:
    a.load()
    print a.surf_name
    actor_movies = a.surf_movies
    for am in actor_movies:
        print '\tStarred in %s' % am.surf_title

print actors[0].serialize('n3')
print '-----------------------------------'
print actors[0].serialize('nt')
print '-----------------------------------'
print actors[0].serialize('json')

print 'done'
print 'Size of store ', session.default_store.size()
```

1.2.5 Creating a Pylons Blog, on *SuRF*

The example is an adaptation of the following example

- http://wiki.pylonshq.com/display/pylonscookbook/Making+a+Pylons+Blog

---

Note: This was tested with *pylons 0.9.7*. To use the latest version of *pylons* update example accordingly.

---

1. Install `pylons`

   ```
   $ easy_install pylons
   ```

2. Create a *pylons* application called *MyBlog*

   ```
   $ cd /home/user/workspace
   $ paster create -t pylons MyBlog
   $ cd MyBlog
   ```

3. The Models and the Data.

   For this example we use the *AllegroGraph* RDF store. See the *Install and Configure AllegroGraph RDF Store* page The default *engine* has been left in, just as in the original example, one can take it out if needed.

   3.1. Edit the *~/MyBlog/development.ini* file and add the following lines

   ```
   [app-main]
   ...
   surf.reader     = allegro_franz
   surf.writer     = allegro_franz
   surf.server     = localhost
   surf.port       = 6789
   surf.catalog    = repositories
   surf.repository = surf_blog
   surf.logging    = true
   surf.clear      = false
   myblog.namespace= http://myblog.com/ns#
   ...
   ```

   3.2. Edit the *~/MyBlog/myblog/config/environment.py* file Add the following lines at the top of the file

```
from surf import *
from myblog.model import *
```

and the following at the end of the `load_environment()` method

```
rdf_store = Store(reader    = config['surf.reader'],
                  writer    = config['surf.writer'],
                  server    = config['surf.server'],
                  port      = config['surf.port'],
                  catalog   = config['surf.catalog'],
                  repository= config['surf.repository'])

if config['surf.clear'] == 'true':
    rdf_store.clear()
print 'SIZE of STORE : ',rdf_store.size()

# the surf session
rdf_session = Session(rdf_store, {})
rdf_session.enable_logging = True if config['surf.logging'] == 'true' else False

# register the namespace
ns.register(myblog=config['myblog.namespace'])

init_model(rdf_session)
```

3.3. Edit the *~/MyBlog/myblog/model/__ init __.py* file

```
from surf import *

def init_model(session):
    """Call me before using any of the tables or classes in the model"""
    global rdf_session
    rdf_session = session

    global Blog
    Blog = rdf_session.get_class(ns.MYBLOG['Blog'])
```

3.4. **Optional** You can edit *~/MyBlog/myblog/websetup.py* to add initial data in the RDF store or just to run maintenance tasks for your *pylons* application, but this is not needed yet

3.5. **Optional** You can setup your application by issuing the following command:

```
$ paster setup-app development.ini
```

4. Putting the script together

4.1. Creating the *blog* controller

```
$ paster controller blog
```

4.2. Edit the *~/MyBlog/myblog/controllers/blog.py* file

```
import logging

from pylons import request, response, session, tmpl_context as c
from pylons.controllers.util import abort, redirect_to

from myblog.lib.base import *
from myblog import model
```

```
log = logging.getLogger(__name__)

class BlogController(BaseController):

    def index(self):
        c.posts = model.Blog.all(limit=5)
        return render("/blog/index.html")
```

4.3. Create the template

```
$ mkdir ~/MyBlog/myblog/templates/blog
```

4.4. Edit the template *~/MyBlog/myblog/templates/blog/index.html*

```
<%inherit file="site.html" />
<%def name="title()">MyBlog Home</%def>

<p>${len(c.posts)} new blog posts!</p>

% for post in c.posts:
<p class="content" style="border-style:solid;border-width:1px">
        <span class="h3"> ${post.dc_title} </span>
        <span class="h4">Posted on: ${post.dc_created} by ${post.sioc_has_creator}</span>
        <br>
          ${post.sioc_content}
</p>
% endfor

<hr/>
<a href="/toolkit/index">Admin</a>
```

For this example the following properties were chosen to describe a blog post in this system, the *sioc:content* describes the content of the post, *sioc:has_author* describes the author, the *dc:created* describes the creation date and the *dc:title* describes the title of the post.

4.5. Edit the *~/MyBlog/myblog/templates/blog/site.html* file

```
<%def name="title()"></%def>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/html4/lo
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>MyBlog: ${self.title()}</title>
    </head>
    <body>
        <h1>${self.title()}</h1>

<!-- *** BEGIN page content *** -->
${self.body()}
<!-- *** END page content *** -->

    </body>
</html>
```

4.6. **Optional** Add the transaction logger to the blog system. Edit the *~/MyBlog/myblog/config/middleware.py* file

at the begining

```
from paste.translogger import TransLogger
```

in the `make_app()` method add the following

```
# CUSTOM MIDDLEWARE HERE
format = ('%(REMOTE_ADDR)s - %(REMOTE_USER)s [%(time)s] '
  '"%(REQUEST_METHOD)s %(REQUEST_URI)s %(HTTP_VERSION)s" '
  '%(status)s %(bytes)s')
app = TransLogger(app, format=format, logger_name="access")
```

4.7. Test the application:

```
$  paster serve --reload development.ini
Starting subprocess with file monitor
01:55:52,596 INFO  [rdflib] version: 2.4.2
surf.plugin allegro_franz reader : franz libraries installed
surf.plugin allegro_franz writer : franz libraries installed
01:55:52,682 INFO  [Store] initializing the store
01:55:52,682 INFO  [Store] registered readers : ['sparql_protocol', 'allegro_franz', 'sesame
01:55:52,683 INFO  [Store] registered writer : ['allegro_franz', 'sesame2']
01:55:52,711 INFO  [Store] store initialized
Starting server in PID 14993.
serving on http://127.0.0.1:5000
```

Test the application on: http://localhost:5000/blog/index, the following should be displayed:

```
MyBlog Home

0 new blog posts!
```

4.8. The home pace. Delete the *~/MyBlog/myblog/public/index.html* file. Edit the *~/MyBlog/myblog/config/routing.py* file

After the *# CUSTOM ROUTES HERE* add this line

```
map.connect('/', controller='blog', action='index')
```

5. Adding a toolkit. The *admin* frontend

5.1. Add the *toolkit* controller

```
$ paster controller toolkit
```

5.2. Create the *toolkit* templates

```
$ mkdir ~/MyBlog/myblog/templates/toolkit
```

edit *~/MyBlog/myblog/templates/toolkit/index.html*

```
<%inherit file="/blog/site.html" />
<%def name="title()">Admin Control Panel</%def>

This is home of the toolkit. <br>
For now you can only
<a href="${h.url_for(controller="toolkit", action="blog_add")}">add</a>
blog posts.
<p>
Later on you'll be able to delete and edit also.
```

edit *~/MyBlog/myblog/templates/toolkit/add.html*

```
<%inherit file="/blog/site.html" />
<%def name="title()">Add Blog Post</%def>

<span class="h3"> Post a Comment </span>
${h.form('/toolkit/blog_add_process')}
<label>Subject: ${h.text('title')}</label><br>
<label>Author: ${h.text('author')}</label><br>
<label>Post Content: ${h.textarea('content')}</label><br>
${h.submit('Submit','Post New Page')}
${h.end_form()}
```

5.3. Change the controller so that it handles the new actions. Edit *~/MyBlog/myblog/controllers/toolkit.py*

```
import datetime
import logging

from pylons import request, response, session, tmpl_context as c
from pylons.controllers.util import abort, redirect_to
from myblog.lib.base import *
from myblog import model
from surf import *

log = logging.getLogger(__name__)

class ToolkitController(BaseController):

    def index(self):
        return render('/toolkit/index.html')

    def blog_add(self):
        return render('/toolkit/add.html')

    def blog_add_process(self):
        # Create a new Blog object and populate it.
        # if you do not specify a subject, one will automatically be generated for you
        # in the surf namespace
        newpost = model.Blog()
        newpost.dc_created = datetime.datetime.now()
        newpost.sioc_content = request.params['content']
        newpost.sioc_has_creator = request.params['author']
        newpost.dc_title = request.params['title']

        # commit the changes - the session tracks Resources automatically
        model.rdf_session.commit()

        # Redirect to the blog home page.
        redirect_to("/")
```

5.4. Edit the *~/MyBlog/myblog/lib/helpers.py* file, add the line in the import section

```
from routes import url_for
from webhelpers.html.tags import *
```

edit the *~/MyBlog/myblog/lib/base.py* file, add the line in the import section

```
import helpers as h
```

6. Thant's it :), Try it out.Test the toolkit interface on:

http://localhost:5000/toolkit/index

1.3 The *Store* and the *Session*

1.3.1 What do `surf.store.Store` and `surf.session.Session` do?

The *Session* establishes all conversations to the backend storage service. Resources use it to load and save their constituting triples. The *Session* keeps a cache of already loaded data, and it uses one or more stores to do actual loading and presistence of data.

The *Store* provides functions for loading and saving data, these are divided into **reader** and **writer** sub-components. *Readers* and *writers* are provided by plugins.

1.3.2 Preparing the *store* and the *session*

The *Store* and the *Session* objects can be instantiated as any regular Python object. Instantiation of *store* and *session* objects is illustrated below:

```
import surf
store = surf.Store(reader = "rdflib", writer = "rdflib")
session = surf.Session(store)
```

the *Store* is configured using its constructor arguments. `reader` and `writer` arguments specify which plugin is to be used for reading and writing RDF data. Possible values (but not limited to) for these two arguments are *sparql_protocol*, *rdflib*, *allegro_franz* and *sesame2*. Plugin-specific configuration options are also specified as constructor argument for Store. In this example, *store* is configured to use the *sparql_protocol* plugin and the address of the **SPARQL HTTP** endpoint is also specified:

```
import surf
store = surf.Store(reader = "sparql_protocol",
                   endpoint = "http://dbpedia.org/sparql")
session = surf.Session(store)
```

It is often convenient to load Store configuration options from file instead of specifying them in code. For example, consider an .ini file with the following contents:

```
[surf]
reader=sparql_protocol
endpoint=http://dbpedia.org/sparql
```

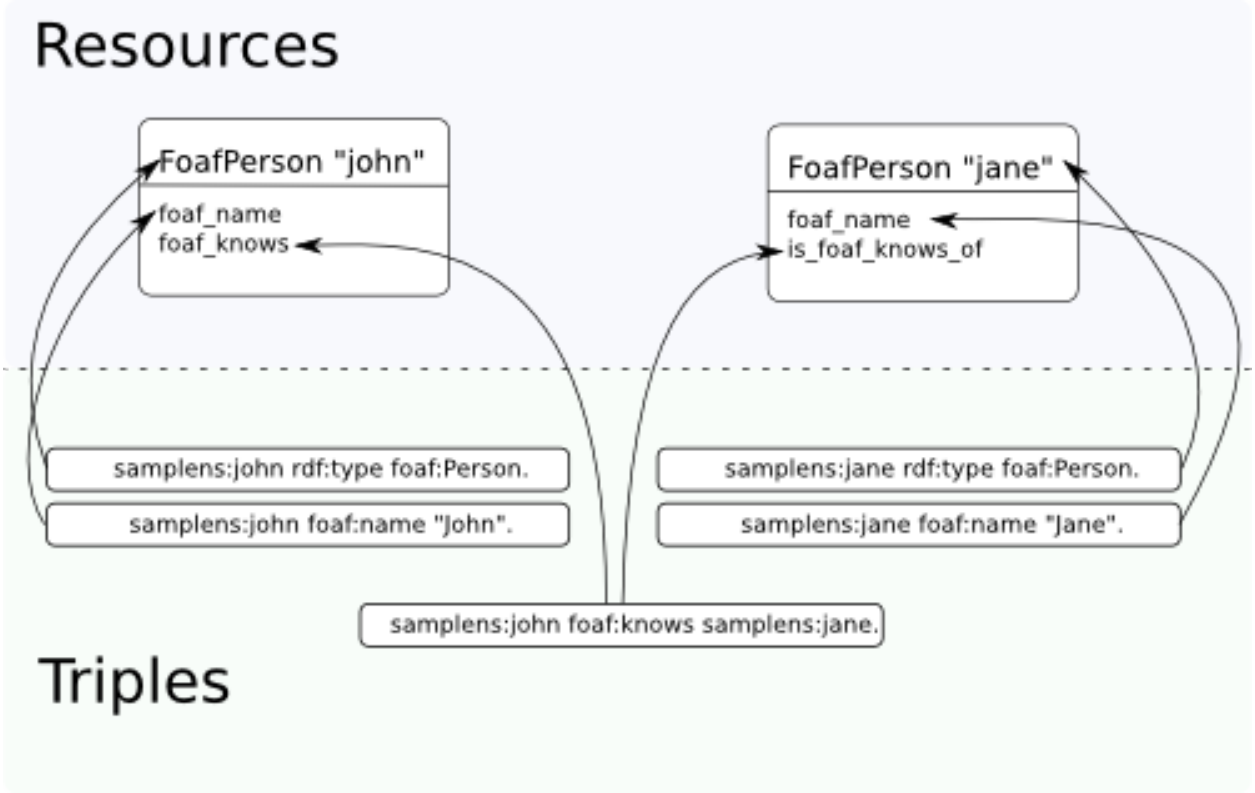
The following snippet loads all configuration keys from the **[surf]** section of the **ini** file and passes them to *Store* constructor:

```
import ConfigParser
import surf

config = ConfigParser.ConfigParser()
config.readfp(open("sample.ini"))
store_params = dict(config.items("surf"))
store = surf.Store(**store_params)
session = surf.Session(store)
```

1.4 Resources and Classes

SuRF `surf.resource.Resource` objects are the core part of *SuRF*. In *SuRF*, *RDF* data is queried, accessed and modified by working with attributes of *Resource* objects. Here's how the *SuRF Resource* maps to the RDF triples conceptual level:



1.4.1 Getting a single Resource object

If type and URI of resource is known, resource can be loaded using session's `surf.session.Session.get_class()` and `surf.session.Session.get_resource()` methods:

```
# Create FoafPerson class:
FoafPerson = session.get_class(surf.ns.FOAF.Person)
# Create instance of FoafPerson class:
john = session.get_resource("http://john.com/me", FoafPerson)
# or simply like this
john = FoafPerson("http://john.com/me")
```

1.4.2 Loading multiple resources

Getting all instances of *FoafPerson* class, in undefined order:

```
>>> FoafPerson = session.get_class(surf.ns.FOAF.Person)
>>> for person in FoafPerson.all():
...     print "Found person:", person.foaf_name.first
```

```
Found person: ...
Found person: ...
```

Getting instances of *FoafPerson* class named "John":

```
>>> FoafPerson = session.get_class(surf.ns.FOAF.Person)
>>> for person in FoafPerson.get_by(foaf_name = "John"):
...     print "Found person:", person.foaf_name.first
Found person: John
```

Getting ordered and limited list of persons:

```
>>> FoafPerson = session.get_class(surf.ns.FOAF.Person)
>>> for person in FoafPerson.all().limit(10).order(surf.ns.FOAF.name):
...     print "Found person:", person.foaf_name.first
Found person: Jane
Found person: John
```

Other modifiers accepted by `all()` and `get_by` are described in `surf.resource.result_proxy` module.

1.4.3 Using resource attributes

A SuRF resource represents a single RDF resource. Its URI is stored in `subject` attribute:

```
>>> FoafPerson = session.get_class(surf.ns.FOAF.Person)
>>> john = session.get_resource("http://john.com/me", FoafPerson)
>>> print john.subject
http://john.com/me
```

RDF triples that describe this resource are available as object attributes. SuRF follows "prefix_predicate" convention for attribute names. These attributes are instances of `surf.resource.value.ResourceValue` class. They are list-like, with some extra convenience functions:

```
>>> # Print all foaf:name values
    >>> print john.foaf_name
    [rdflib.Literal(u'John')]

    >>> # Print first foaf:name value or None if there aren't any:
    >>> print john.foaf_name.first
    John

    >>> # Print first foaf:name value or raise exception if there aren't any or
    >>> # there are more than one:
    >>> print john.foaf_nonexistant_predicate.one
    Traceback (most recent call last):
    ...
    NoResultFound: list is empty
```

RDF triples that have resource as object, are available as "inverse" attributes, they follow "is_prefix_predicate_of" convention:

```
>>> # Print all persons that know john
>>> print john.is_foaf_knows_of
[<surf.session.FoafPerson object at ...>]
```

Alternatively, dictionary-style attribute access can be used. It is useful in cases where "prefix_predicate" naming convention would yield attribute names that are not valid in Python, like "vcard_postal-code". It can also be used for easy iterating over a list of attributes:

```
>>> for attr in ["name", "surname"]: print john["foaf_%s" % attr].first
John
Smith

>>> # URIRefs are also accepted as dictionary keys:
>>> for attr in ["name", "surname"]: print john[surf.ns.FOAF[attr]].first
John
Smith
```

Attributes can be used as starting points for more involved querying:

```
>>> # Get first item from ordered list of all friends named "Jane":
>>> john.foaf_knows.get_by(foaf_name = "Jane").order().first()
<surf.session.FoafPerson object at ...>
```

Modifiers accepted by attributes are described in `surf.resource.result_proxy` module.

1.4.4 Saving, deleting resources

Saving a resource:

```
resource.save()
```

Deleting a resource:

```
resource.remove()
```

SuRF will allow instantiate resource with any URI and type, regardless of whether such resource is actually present in triple store. To tell if instantiated resource is present in triple store use `surf.resource.Resource.is_present()` method:

```
>>> resource = session.get_resource("http://nonexistant-uri", surf.ns.OWL.Thing)
>>> resource.is_present()
False
```

1.4.5 Extending SuRF resource classes

SuRF Resource objects are all instances of `surf.resource.Resource`. It is possible to specify additional classes that resources of particular RDF type should subclass. This lets applications add custom logic to resource classes based on their type. The mapping is defined at session level by populating `mapping` dictionary in session object:

```
class MyPerson(object):
        """ Some custom logic for foaf:Person resources. """

        def get_friends_count(self):
                return len(self.foaf_knows)


session.mapping[surf.ns.FOAF.Person] = MyPerson

# Now let's test the mapping
john = session.get_resource("http://example/john", surf.ns.FOAF.Person)

# Is `john` an instance of surf.Resource?
print isinstance(john, surf.Resource)
# outputs: True
```

```
# Is 'john' an instance of MyPerson?
print isinstance(john, MyPerson)
# outputs: True

# Try the custom 'get_friends_count' method:
print john.get_friends_count()
# outputs: 0
```

1.5 Queries

SuRF aims to integrate **RDF** with the *object-oriented* paradigm so that manual writing and execution of **SPARQL** queries is seldom needed. Resources and classes provide a higher level of abstraction than queries do and they should cover the most common use cases.

1.5.1 Executing arbitrary SPARQL queries

It is still possible to execute arbitrary queries in the cases where this is needed. The `surf.store.Store` class provides the method: `surf.store.Store.execute_sparql()` which accepts the query as a string. This method will return raw results, and *SuRF* will make no attempt to represent returned data as resource objects.

```
>>> import surf
>>> from surf.rdf import URIRef
>>> sess = surf.Session(surf.Store(reader="rdflib", writer="rdflib"))
>>> sess.default_store.add_triple(URIRef("http://s"), URIRef("http://p"), "value!")

>>> sess.default_store.execute_sparql("SELECT ?s ?p ?o WHERE { ?s ?p ?o }")
<rdflib.sparql.QueryResult.SPARQLQueryResult object at ...>

>>> list(sess.default_store.execute_sparql("SELECT ?s ?p ?o WHERE { ?s ?p ?o }"))
[(rdflib.URIRef('http://s'), rdflib.URIRef('http://p'), 'value!')]
```

1.5.2 Constructing queries in a programmatic way

SuRF also provides utilities for programmatic construction of dynamic **SPARQL** queries in the `surf.query` module. Using them can sometimes result in cleaner code than constructing queries by string concatenation. Here's an example on how to use the tools available in the `surf.query` module:

```
>>> import surf
>>> from surf.query import a, select
>>> query = select("?s", "?src")
>>> query.named_group("?src", ("?s", a, surf.ns.FOAF['Person']))
>>> print unicode(query)
SELECT  ?s ?src  WHERE {  GRAPH ?src {  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http:/
```

1.6 Data Access Methods

1.6.1 The *allegro_franz* Plugin

Table 1.1: Input Parameters

Parameter	Default Value	Description
server	*.localhost*	the location of the *AllegroGraph* RDF server
port	*6789*	the port *AllegroGraph* is running on
catalog	*None*	the catalog to use
repository	*None*	the repository to use

the parameters are passed as key-value arguments to the `surf.store.Store` class

```
s = Store(  reader      = 'allegro_franz',
            writer      = 'allegro_franz',
            server      = 'localhost',
            port        = 6789,
            catalog     = 'repositories',
            repository  = 'test_surf')
```

Setting up *AllegroGraph* RDF Store

Install and Configure *AllegroGraph* RDF Store

Download *AllegroGraph* from here http://www.franz.com/downloads/clp/ag_survey after you complete the Franz online survey. The free version of *AllegroGraph* is limited to 50.000.000 RDF triples.

Installing on Windows *AllegroGraph* is installed as a *windows service*. After the installation if complete one must proceed to configure the RDF store

1. Create a folder on disk where the store *repositories* will reside, say D:\repositories
2. Open and edit the *[AllegroGraph installation directory]\agraph.cfg* file and change it accordingly

```
;; This file contains the configuration options for AllegroGraph.
;; Please refer to the installation documentation for the
;; AllegroGraph server for information on valid values for these options.
;;
;; Comments start with a semicolon (;).
(
;; Please do not change the following line:
(:agraph-server-config 5)
;; User-settable options start here:
:direct nil
:new-http-port 6789
:new-http-auth nil
:new-http-catalog ("D:/repositories")
:http-port -1
:http-init-file nil
:http-only nil
:idle-life 86400
:eval-in-server-file nil
:pid-file "sys:agraph.pid"
:client-prolog nil
```

```
:index -1
:init-file "sys:aginit.cl"
:lease -1
:limit -1
:log-file "sys:agraph.log"
:no-direct nil
:no-java nil
:port 4567
:port2 4568
:res -1
:repl-port nil
:standalone t
:timeout 60
:error-log nil
:users 50
:verbose t
)
;;END OF CONFIG
```

the location of the repositories folder can be any, so can the port

3. Copy the *[AllegroGraph installation directory]\python\franz* directory to *[Python installation directory]\lib\site-packages* and install the required python libs as requested in the documentation

4. Update *AllegroGraph* and restart the service

Installing on Linux Extract *AllegroGraph* to a location of your choosing

1. Create a folder on disk where the store *repositories* will reside, say /home/user/repositories

2. Open and edit the *[AllegroGraph installation directory]/agraph.cfg* file and change it accordingly

```
;; This file contains the configuration options for AllegroGraph.
;; Please refer to the installation documentation for the
;; AllegroGraph server for information on valid values for these options.
;;
;; Comments start with a semicolon (;).
(
;; Please do not change the following line:
(:agraph-server-config 5)
;; User-settable options start here:
:direct nil
:new-http-port 6789
:new-http-auth nil
:new-http-catalog ("/home/user/repositories")
:http-port -1
:http-init-file nil
:http-only nil
:idle-life 86400
:eval-in-server-file nil
:pid-file "sys:agraph.pid"
:client-prolog nil
:index -1
:init-file "sys:aginit.cl"
:lease -1
:limit -1
:log-file "sys:agraph.log"
:no-direct nil
:no-java nil
```

```
:port 4567
:port2 4568
:res -1
:repl-port nil
:standalone t
:timeout 60
:error-log nil
:users 50
:verbose t
)
;;END OF CONFIG
```

the location of the repositories folder can be any, so can the port

3. Copy the *[AllegroGraph installation directory]/python/franz* directory to *[Python installation directory]/site-packages* and install the required python libs as requested in the documentation

4. Update *AllegroGraph* and restart the service

1.6.2 The *rdflib* Plugin

Table 1.2: Input Parameters

Parameter	Default Value	Description
rdflib_store	*IOMemory*	Default rdflib storage backend to use
rdflib_identifier	*None*	Identifier to use for default graph

The parameters are passed as key-value arguments to the `surf.store.Store` class.

```
s = Store(  reader            =   "rdflib",
            writer            =   "rdflib",
            rdflib_store      =   "IOMemory",
            rdflib_identifier =   URIRef("http://my_graph_uri"))
```

1.6.3 The *sesame2* Plugin

Table 1.3: Input Parameters

Parameter	Default Value	Description
server	*localhost*	the location of the *AllegroGraph* RDF server
port	*6789*	the port *AllegroGraph* is running on
repository	*None*	the repository to use
root_path	*/sesame*	the sesame http api root path pf the server
repository_path		the location on disk of the directory holding the repository
use_allegro_extensions	*False*	whether to use AllegroGraph Extensions to Sesame2 HTTP protocol. Set this to true if the repository you are accessing over Sesame2 HTTP protocol is AllegroGraph.

the parameters are passed as key-value arguments to the `surf.store.Store` class

```
s = Store(  reader          = 'sesame2',
            writer          = 'sesame2',
            server          = 'localhost',
```

```
            port            = 6789,
            repository      = 'test_surf',
            root_path       = '/sesame',
            repository_path =   r'D:\repositories')
```

1.6.4 The *sparql_protocol* Plugin

sparql_protocol plugin reads data from SPARQL endpoints. It also implements writing to endpoints using SPARQL-Update language. This plugin is known to work with endpoints supplied by OpenLink Virtuoso and 4store. It currently cannot access endpoints that require authorization.

SPARQLWrapper library is used for actually making requests and converting data from and to Python structures.

Table 1.4: Initialization Parameters

Parameter	Default Value	Description
endpoint	*None*	Address of SPARQL HTTP endpoint.
default_context	*None*	The default context (graph) to be queried against (this is useful in particular for the Virtuoso RDF store).
combine_queries	*None*	whether multiple SPARUL queries can be sent in one request
use_subqueries	*None*	whether use of SPARQL 1.1 subqueries and SELECT expressions is allowed (whether SPARQL endpoint supports that)
use_keepalive	*False*	whether to use HTTP 1.1 keep-alive connections.

The parameters are passed as key-value arguments to the `surf.store.Store` class:

```
s = Store(  reader          =   "sparql_protocol",
            writer          =   "sparql_protocol",
            endpoint        =   "http://dbpedia.org/sparql",
            default_graph   =   "http://dbpedia.org")
```

1.6.5 Setting up *OpenLink Virtuoso* RDF Store

Install and Configure *OpenLink Virtuoso* RDF Store

Installing on Windows

The instructions and documentation on how to run *SuRF* on top of *OpenLink Virtuoso* were contributed by Peteris Caune further updates and information can be read here .

1. Download virtuoso-opensource-win32-5.0.11.zip or a more recent version (unavailable at the writing time of this document)

Note: For the purpose of this example version 5.0.11 of *virtuoso* was used, any other version can be used instead.

---

2. Extract it to c:\virtuoso
3. Add c:\virtuoso to system *PATH*

3.1. **Optional** Adjust c:\virtuoso\database\virtuoso.ini as needed ou can change port number for Virtuoso's web interface, how much memory it uses, which plugins it loads and so forth, documentation here.

4. Execute from shell:

```
$ cd c:\virtuoso\database
$ virtuoso-t -f virtuoso.ini
```

Note: the *-f* flag sets *virtuoso* to run in the foreground

---

5. Go explore web frontend at http://localhost:8890. Default *username/password* for administrator is *dba/dba*.

6. To communicate with Virtuoso, SuRF will use it's SPARQL endpoint at http://localhost:8890/sparql. By default this endpoint has no write rights. To grant these rights, launch *isql* utility from shell and execute this line in it:

```
grant SPARQL_UPDATE to "SPARQL";
```

Such a setup configuration is fine for development and testing, but having a public writable *SPARQL* endpoint on production system is probably not a good idea.

1.7 Using *SuRF* with RDF triple stores

1.7.1 Install and Configure *OpenLink Virtuoso* RDF Store

Installing on Windows

The instructions and documentation on how to run *SuRF* on top of *OpenLink Virtuoso* were contributed by Peteris Caune further updates and information can be read here .

1. Download virtuoso-opensource-win32-5.0.11.zip or a more recent version (unavailable at the writing time of this document)

Note: For the purpose of this example version 5.0.11 of *virtuoso* was used, any other version can be used instead.

---

2. Extract it to c:\virtuoso

3. Add c:\virtuoso to system *PATH*

3.1. **Optional** Adjust c:\virtuoso\database\virtuoso.ini as needed ou can change port number for Virtuoso's web interface, how much memory it uses, which plugins it loads and so forth, documentation here.

4. Execute from shell:

```
$ cd c:\virtuoso\database
$ virtuoso-t -f virtuoso.ini
```

Note: the *-f* flag sets *virtuoso* to run in the foreground

---

5. Go explore web frontend at http://localhost:8890. Default *username/password* for administrator is *dba/dba*.

6. To communicate with Virtuoso, SuRF will use it's SPARQL endpoint at http://localhost:8890/sparql. By default this endpoint has no write rights. To grant these rights, launch *isql* utility from shell and execute this line in it:

```
grant SPARQL_UPDATE to "SPARQL";
```

Such a setup configuration is fine for development and testing, but having a public writable *SPARQL* endpoint on production system is probably not a good idea.

1.7.2 Install and Configure *AllegroGraph* RDF Store

Download *AllegroGraph* from here http://www.franz.com/downloads/clp/ag_survey after you complete the Franz online survey. The free version of *AllegroGraph* is limited to 50.000.000 RDF triples.

Installing on Windows

AllegroGraph is installed as a *windows service*. After the installation if complete one must proceed to configure the RDF store

1. Create a folder on disk where the store *repositories* will reside, say D:\repositories

2. Open and edit the *[AllegroGraph installation directory]\agraph.cfg* file and change it accordingly

```
;; This file contains the configuration options for AllegroGraph.
;; Please refer to the installation documentation for the
;; AllegroGraph server for information on valid values for these options.
;;
;; Comments start with a semicolon (;).
(
;; Please do not change the following line:
(:agraph-server-config 5)
;; User-settable options start here:
:direct nil
:new-http-port 6789
:new-http-auth nil
:new-http-catalog ("D:/repositories")
:http-port -1
:http-init-file nil
:http-only nil
:idle-life 86400
:eval-in-server-file nil
:pid-file "sys:agraph.pid"
:client-prolog nil
:index -1
:init-file "sys:aginit.cl"
:lease -1
:limit -1
:log-file "sys:agraph.log"
:no-direct nil
:no-java nil
:port 4567
:port2 4568
:res -1
:repl-port nil
```

```
:standalone t
:timeout 60
:error-log nil
:users 50
:verbose t
)
;;END OF CONFIG
```

the location of the repositories folder can be any, so can the port

3. Copy the *[AllegroGraph installation directory]\python\franz* directory to *[Python installation directory]\lib\site-packages* and install the required python libs as requested in the documentation

4. Update *AllegroGraph* and restart the service

Installing on Linux

Extract *AllegroGraph* to a location of your choosing

1. Create a folder on disk where the store *repositories* will reside, say /home/user/repositories

2. Open and edit the *[AllegroGraph installation directory]/agraph.cfg* file and change it accordingly

```
;; This file contains the configuration options for AllegroGraph.
;; Please refer to the installation documentation for the
;; AllegroGraph server for information on valid values for these options.
;;
;; Comments start with a semicolon (;).
(
;; Please do not change the following line:
(:agraph-server-config 5)
;; User-settable options start here:
:direct nil
:new-http-port 6789
:new-http-auth nil
:new-http-catalog ("/home/user/repositories")
:http-port -1
:http-init-file nil
:http-only nil
:idle-life 86400
:eval-in-server-file nil
:pid-file "sys:agraph.pid"
:client-prolog nil
:index -1
:init-file "sys:aginit.cl"
:lease -1
:limit -1
:log-file "sys:agraph.log"
:no-direct nil
:no-java nil
:port 4567
:port2 4568
:res -1
:repl-port nil
:standalone t
:timeout 60
:error-log nil
:users 50
:verbose t
```

```
)
;;END OF CONFIG
```

the location of the repositories folder can be any, so can the port

3. Copy the *[AllegroGraph installation directory]/python/franz* directory to *[Python installation directory]/site-packages* and install the required python libs as requested in the documentation

4. Update *AllegroGraph* and restart the service

CHAPTER

TWO

API REFERENCE

2.1 The surf.exc Module

Module for SuRF exceptions.

exception surf.exc.**CardinalityException**
Bases: exceptions.Exception

Raised when list length != 1.

Subclasses of this exception are raised by surf.resource.result_proxy.ResultProxy.one() and surf.resource.value.ResultValue.get_one().

exception surf.exc.**MultipleResultsFound**
Bases: surf.exc.CardinalityException

Raised when list length > 1.

This exception is raised by surf.resource.result_proxy.ResultProxy.one() and surf.resource.value.ResultValue.get_one().

exception surf.exc.**NoResultFound**
Bases: surf.exc.CardinalityException

Raised when list length == 0.

This exception is raised by surf.resource.result_proxy.ResultProxy.one() and surf.resource.value.ResultValue.get_one().

2.2 The surf.namespace Module

surf.namespace.**all**()
Return all the namespaces registered as a dict.

surf.namespace.**base**(*property*)
Return the base part of a URI, *property* is a string denoting a URI.

```
>>> print ns.base('http://sometest.ns/ns#symbol')
http://sometest.ns/ns#
```

surf.namespace.**get_namespace**(*base*)
Return the *namespace* short hand notation and the URI based on the URI *base*.

The namespace is a *rdf.namespace.Namespace*

```
>>> key, namespace = ns.get_namespace('http://sometest.ns/ns#')
>>> print key, namespace
TEST, http://sometest.ns/ns#
```

surf.namespace.**get_namespace_url**(*prefix*)
Return the *namespace* URI registered under the specified *prefix*

```
>>> url = ns.get_namespace_url('TEST')
>>> print url
http://sometest.ns/ns#
```

surf.namespace.**get_prefix**(*uri*)
The inverse function of *get_namespace_url(prefix)*, return the *prefix* of a *namespace* based on its URI.

```
>>> name = ns.get_prefix(Namespace('http://sometest.ns/ns#'))
>>> # true, if one registered the uri to the "test" prefix beforehand
>>> print name
TEST
```

surf.namespace.**register**(***namespaces*)
Register a namespace with a shorthand notation with the *namespace* manager. The arguments are passed in as key-value pairs.

```
>>> ns.register(test='http://sometest.ns/ns#')
>>> print ns.TEST
http://sometest.ns/ns#
```

surf.namespace.**register_fallback**(*namespace*)
Register a fallback namespace to use when creating resource without specifying subject.

```
>>> ns.register_fallback('http://example.com/fallback#')
>>> Person = session.get_class(ns.FOAF.Person)
>>> p = Person()
>>> p.subject
http://example.com/fallback#093d460a-a768-49a9-8813-aa5b321d94a8
```

surf.namespace.**symbol**(*property*)
Return the part of a URI after the last **/** or **#**, *property* is a string denoting a URI

```
>>> print ns.symbol('http://sometest.ns/ns#symbol')
symbol
```

2.2.1 Registered general purpose namespaces

The description of each registered *namespace* was collected from the respective URL describing the ontology / vocabulary

surf.namespace.**XMLNS**
http://www.w3.org/XML/1998/namespace

The "xml:" Namespace

surf.namespace.**SKOS**
http://www.w3.org/2004/02/skos/core#

SKOS Simple Knowledge Organization System Namespace Document

surf.namespace.**XSD**
http://www.w3.org/2001/XMLSchema#

XML Schema

surf.namespace.**OWL**
http://www.w3.org/2002/07/owl#

The Web Ontology Language, This file specifies in RDF Schema format the built-in classes and properties that together form the basis of the RDF/XML syntax of OWL Full, OWL DL and OWL Lite. We do not expect people to import this file explicitly into their ontology. People that do import this file should expect their ontology to be an OWL Full ontology.

surf.namespace.**VS**
http://www.w3.org/2003/06/sw-vocab-status/ns#

SemWeb Vocab Status ontology, An RDF vocabulary for relating SW vocabulary terms to their status.

surf.namespace.**WOT**
http://xmlns.com/wot/0.1/

Web Of Trust RDF Ontology

surf.namespace.**DC**
http://purl.org/dc/elements/1.1/

DCMI Namespace for the Dublin Core Metadata Element Set, Version 1.1

surf.namespace.**IBIS**
http://purl.org/ibis#

IBIS Vocabulary, Issue-Based Information Systems (IBIS) is a collaborative problem analysis and solving technique.

surf.namespace.**SIOC**
http://rdfs.org/sioc/ns#

SIOC (Semantically-Interlinked Online Communities) is an ontology for describing the information in online communities.

surf.namespace.**SIOC_TYPES**
http://rdfs.org/sioc/types#

Extends the SIOC Core Ontology (Semantically-Interlinked Online Communities) by defining subclasses and subproperties of SIOC terms.

surf.namespace.**SIOC_SERVICES**
http://rdfs.org/sioc/services#

Extends the SIOC Core Ontology (Semantically-Interlinked Online Communities) by defining basic information on community-related web services.

surf.namespace.**ATOM**
http://atomowl.org/ontologies/atomrdf#

The ATOM OWL vocabulary

surf.namespace.**EXIF**
http://www.w3.org/2003/12/exif/ns/

Vocabulary to describe an Exif format picture data. All Exif 2.2 tags are defined as RDF properties, as well as several terms to help this schema.

surf.namespace.**ANNOTEA**
http://www.w3.org/2002/01/bookmark#

The Annotea Bookmark Schema, describing properties used to define instances of bookmarks, topics, and shortcuts.

surf.namespace.**RESUME**
http://captsolo.net/semweb/resume/cv.rdfs#

the Resume RDF schema

surf.namespace.**REVIEW**
http://www.isi.edu/webscripter/communityreview/abstract-review-o#

The upper ontology for all semantic web community reviews

surf.namespace.**CALENDAR**
http://www.w3.org/2002/12/cal/icaltzd#

W3C Calendar vocabulary

surf.namespace.**ANNOTATION**
http://www.w3.org/2000/10/annotation-ns#

Annotea Annotation Schema

surf.namespace.**DOAP**
http://usefulinc.com/ns/doap#

Description of a Project (DOAP) vocabulary, The Description of a Project (DOAP) vocabulary, described using W3C RDF Schema and the Web Ontology Language.

surf.namespace.**FOAF**
http://xmlns.com/foaf/0.1/

FOAF Vocabulary Specification. FOAF is a collaborative effort amongst Semantic Web developers on the FOAF (foaf-dev@lists.foaf-project.org) mailing list. The name 'FOAF' is derived from traditional internet usage, an acronym for "Friend of a Friend"

surf.namespace.**GR**
http://purl.org/goodrelations/v1#

GoodRelations is a standardized vocabulary for product, price, and company data that can (1) be embedded into existing static and dynamic Web pages and that (2) can be processed by other computers. This increases the visibility of your products and services in the latest generation of search engines, recommender systems, and other novel applications.

surf.namespace.**WIKIONT**
http://sw.deri.org/2005/04/wikipedia/wikiont.owl

WIKI vocabulary

surf.namespace.**WORDNET**
http://xmlns.com/wordnet/1.6/

Wordnet vocabulary

surf.namespace.**GEO**
http://www.w3.org/2003/01/geo/wgs84_pos#

WGS84 Geo Positioning: an RDF vocabulary, A vocabulary for representing latitude, longitude and altitude information in the WGS84 geodetic reference datum. Version $Id: wgs84_pos.rdf,v 1.22 2009/04/20 15:00:30 timbl Exp $. See http://www.w3.org/2003/01/geo/ for more details.

surf.namespace.**PIM**
http://www.w3.org/2000/10/swap/pim/contact#

PIM vocabulary

surf.namespace.**IMDB**
http://www.csd.abdn.ac.uk/~ggrimnes/dev/imdb/IMDB#

The Internet Movie Database vocabulary, IMDB

surf.namespace.**CONTACT**
http://www.w3.org/2000/10/swap/pim/contact#

The PIM CONTACT vocabulary

surf.namespace.**MARCONT**
http://www.marcont.org/ontology#

MarcOnt Ontology Specification, The goal of MarcOnt bibliographic ontology is to provide a uniform bibliographic description format. It should capture concepts from existing formats such as Bibtex, Dublin Core, MARC21.

surf.namespace.**XFOAF**
http://www.foafrealm.org/xfoaf/0.1/

FOAFRealm Ontology Specification, Proposed FOAFRealm (Friend-of-a-Friend Realm) system allows to take advantage of social networks and FOAF profiles in user profile management systems. However, the FOAF standard must be enriched with new concepts and properties that are described in this document. The enriched version is called FOAFRealm.

surf.namespace.**JDL_STRUCTURE**
http://www.jeromedl.org/structure#

JeromeDL Ontology Specification, The structure ontology is used at the bottom layer in JeromeDL. It is used to handle typical tasks required from a digital objects repository, that is, it keeps track of physical representation of resources, their structure and provenance. The structure ontology provides means for a flexible and extendable electronic representation of objects. Such flexibility is especially significant in expressing relations to other resources

surf.namespace.**JONTO_PKT**
http://www.corrib.org/jonto/pkt#

JONTO PKT (JeromeDL) vocabulary

surf.namespace.**JONTO_DDC**
http://www.corrib.org/jonto/ddc#

JONTO DDC (JeromeDL) vocabulary

surf.namespace.**CORRIB_TAX**
http://jonto.corrib.org/taxonomies#

CORRIB Taxonomies (JeromeDL) vocabulary

surf.namespace.**SERENITY3**
http://serenity.deri.org/imdb#

The SERENITY vocabulary

surf.namespace.**IDEAS**
http://protege.stanford.edu/rdf

The IDEAS vocabulary, PROTEGE

surf.namespace.**BIBO**
http://purl.org/ontology/bibo/

The Bibliographic Ontology, The Bibliographic Ontology describe bibliographic things on the semantic Web in RDF. This ontology can be used as a citation ontology, as a document classification ontology, or simply as a way

to describe any kind of document in RDF. It has been inspired by many existing document description metadata formats, and can be used as a common ground for converting other bibliographic data sources.

surf.namespace.**FRBR**
http://purl.org/vocab/frbr/core#

Expression of Core FRBR Concepts in RDF, This vocabulary is an expression in RDF of the concepts and relations described in the IFLA report on the Functional Requirements for Bibliographic Records (FRBR).

surf.namespace.**MO**
http://purl.org/ontology/mo/

Music Ontology Specification, The Music Ontology Specification provides main concepts and properties fo describing music (i.e. artists, albums, tracks, but also performances, arrangements, etc.) on the Semantic Web. This document contains a detailed description of the Music Ontology.

surf.namespace.**VCARD**
http://nwalsh.com/rdf/vCard#

This ontology attempts to model a subset of vCards in RDF using modern (circa 2005) RDF best practices. The subset selected is the same subset that the microformats community has adopted for use in hCard

surf.namespace.**VANN**
http://purl.org/vocab/vann/

VANN: A vocabulary for annotating vocabulary descriptions, This document describes a vocabulary for annotating descriptions of vocabularies with examples and usage notes.

surf.namespace.**EVENT**
http://purl.org/NET/c4dm/event.owl#

The Event Ontology, This document describes the Event ontology developed in the Centre for Digital Music in Queen Mary, University of London.

surf.namespace.**VS**
http://www.w3.org/2003/06/sw-vocab-status/ns#

SemWeb Vocab Status ontology, An RDF vocabulary for relating SW vocabulary terms to their status.

surf.namespace.**TIME**
http://www.w3.org/2006/time#

An OWL Ontology of Time (OWL-Time), A paper, "An Ontology of Time for the Semantic Web", that explains in detail about a first-order logic axiomatization of OWL-Time can be found at:

- http://www.isi.edu/~pan/time/pub/hobbs-pan-TALIP04.pdf

More materials about OWL-Time:

- http://www.isi.edu/~pan/OWL-Time.html
- http://www.w3.org/TR/owl-time

surf.namespace.**WGS84_POS**
http://www.w3.org/2003/01/geo/wgs84_pos#

WGS84 Geo Positioning: an RDF vocabulary, A vocabulary for representing latitude, longitude and altitude information in the WGS84 geodetic reference datum. See http://www.w3.org/2003/01/geo/ for more details.

surf.namespace.**BIBO_ROLES**
http://purl.org/ontology/bibo/roles/

The BIBO Roles vocabulary

surf.namespace.**BIBO_DEGREES**
http://purl.org/ontology/bibo/degrees/

The BIBO Degrees vocabulary

surf.namespace.**BIBO_EVENTS**
http://purl.org/ontology/bibo/events/

The BIBO Events vocabulary

surf.namespace.**BIBO_STATUS**
http://purl.org/ontology/bibo/status/

The BIBO Status vocabulary

surf.namespace.**FRESNEL**
http://www.w3.org/2004/09/fresnel#

Fresnel Lens and Format Core Vocabulary, OWL Full vocabulary for defining lenses and formats on RDF models.

surf.namespace.**DCTERMS**
http://purl.org/dc/terms/

DCMI Namespace for metadata terms in the http://purl.org/dc/terms/ namespace

surf.namespace.**DBPEDIA**
http://dbpedia.org/property/

DBpedia, An Entity in Data Space: dbpedia.org

surf.namespace.**YAGO**
http://dbpedia.org/class/yago/

DBpedia YAGO Classes, An Entity in Data Space: dbpedia.org

surf.namespace.**LUBM**
http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#

Univ-bench Ontology, An university ontology for benchmark tests

surf.namespace.**DBLP**
http://www4.wiwiss.fu-berlin.de/dblp/terms.rdf#

DBLP vocabulary

surf.namespace.**FTI**
http://franz.com/ns/allegrograph/2.2/textindex/

Franz AllegroGraph, namespace for Free Text Indexing, used by AllegroGraph to specify predicates that can be used in SPARQL queries to perform free text indexing

surf.namespace.**SURF**
http://code.google.com/p/surfrdf/

The SuRF namespace is used internally by `surf` to generate unique subjects for *resources* if a subject is not provided

2.3 The `surf.rdf` module

Helper module that conditionally loads *rdflib* classes and functionality. The following classes are exposed:

- `BNode`

- `ClosedNamespace`
- `ConjunctiveGraph`
- `Graph`
- `Literal`
- `Namespace`
- *RDF*
- *RDFS*
- `URIRef`

2.4 The `surf.plugin` Module

2.4.1 Contents

The `surf.plugin.manager` Module

exception `surf.plugin.manager.`**`PluginNotFoundException`**
Bases: `exceptions.Exception`

Raised when the required Plugin is not found

`surf.plugin.manager.`**`add_plugin_path`**(*plugin_path*)
Loads plugins from *path*. Method can be called multiple times, with different locations. (Plugins are loaded only once).

`surf.plugin.manager.`**`load_plugins`**(*reload=False*)
Call this method to load the plugins into the manager. The method is called by default when a `surf.store.Store` is instantiated. To cause a reload, call the method with *reload* set to *True*

The `surf.plugin.reader` Module

class `surf.plugin.reader.`**`RDFReader`**(**args*, ***kwargs*)
Bases: `surf.plugin.Plugin`

Super class for all surf Reader plugins.

`close`()
Close the *plugin* and free any resources it may hold.

`concept`(*resource*)
Return the *concept* URI of the following *resource*.

resource can be a *string* or a *URIRef*.

`enable_logging`(*enable=True*)
Enables or disable *logging* for the current *plugin*.

`get`(*resource*, *attribute*, *direct*)
Return the *value(s)* of the corresponding *attribute*.

If `direct` is *False* then the subject of the `resource` is considered the object of the query.

instances_by_attribute(*resource*, *attributes*, *direct*, *context*)
Return all *URIs* that are instances of `resource` and have the specified *attributes*.

If `direct` is *False*, than the subject of the `resource` is considered the object of the query.

is_enable_logging()
True if *logging* is enabled.

is_present(*resource*)
Return *True* if the `resource` is present in the *store*.

load(*resource*, *direct*)
Fully load the `resource` from the *store*.

This method returns all statements about the *resource*.

If `direct` is *False*, then the subject of the `resource` is considered the object of the query

The surf.plugin.query_reader Module

class surf.plugin.query_reader.**RDFQueryReader**(**args*, ***kwargs*)
Bases: surf.plugin.reader.RDFReader

Super class for SuRF Reader plugins that wrap queryable *stores*.

close()
Close the *plugin* and free any resources it may hold.

concept(*resource*)
Return the *concept* URI of the following *resource*.

resource can be a *string* or a *URIRef*.

convert(*query_result*, **keys*)
Convert the results from the query to a multilevel dictionary.

This method is used by the surf.resource.Resource class.

enable_logging(*enable=True*)
Enables or disable *logging* for the current *plugin*.

execute(*query*)
Execute a *query* of type surf.query.Query.

get(*resource*, *attribute*, *direct*)
Return the *value(s)* of the corresponding *attribute*.

If `direct` is *False* then the subject of the `resource` is considered the object of the query.

instances_by_attribute(*resource*, *attributes*, *direct*, *context*)
Return all *URIs* that are instances of `resource` and have the specified *attributes*.

If `direct` is *False*, than the subject of the `resource` is considered the object of the query.

is_enable_logging()
True if *logging* is enabled.

is_present(*resource*)
Return *True* if the `resource` is present in the *store*.

load(*resource*, *direct*)
Fully load the `resource` from the *store*.

This method returns all statements about the *resource*.

If `direct` is *False*, then the subject of the `resource` is considered the object of the query

`surf.plugin.query_reader.`**`query_Ask`**(*subject*, *context*)
Construct `surf.query.Query` of type **ASK**.

`surf.plugin.query_reader.`**`query_Concept`**(*subject*)
Construct `surf.query.Query` with *?c* as the unknown.

`surf.plugin.query_reader.`**`query_P_S`**(*c*, *p*, *direct*, *context*)
Construct `surf.query.Query` with *?s* and *?c* as unknowns.

`surf.plugin.query_reader.`**`query_S`**(*s*, *direct*, *context*)
Construct `surf.query.Query` with *?p*, *?v* and *?c* as unknowns.

`surf.plugin.query_reader.`**`query_SP`**(*s*, *p*, *direct*, *context*)
Construct `surf.query.Query` with *?v* and *?c* as unknowns.

The `surf.plugin.writer` Module

class `surf.plugin.writer.`**`RDFWriter`**(*reader*, **args*, ***kwargs*)
Bases: `surf.plugin.Plugin`

Super class for all surf Writer plugins.

`add_triple`(*s=None*, *p=None*, *o=None*, *context=None*)
Add a triple to the *store*, in the specified `context`.

None can be used as a wildcard.

`clear`(*context=None*)
Remove all triples from the *store*.

If `context` is specified, only the specified context will be cleared.

`close`()
Close the *plugin*.

`enable_logging`(*enable=True*)
Enables or disable *logging* for the current *plugin*.

`index_triples`(***kwargs*)
Perform *index* of the *triples* if such functionality is present.

Return *True* if operation successful.

`is_enable_logging`()
True if *logging* is enabled.

`load_triples`(***kwargs*)
Load *triples* from supported *sources* if such functionality is present.

Return *True* if operation successful.

`remove`(**resources*, ***kwargs*)
Completely remove the `*resources` from the *store*.

`remove_triple`(*s=None*, *p=None*, *o=None*, *context=None*)
Remove a triple from the *store*, from the specified `context`.

None can be used as a wildcard.

`save`(**resources*)
Replace the `*resources` in store with their current state.

set_triple(*s=None*, *p=None*, *o=None*, *context=None*)
Replace a triple in the *store* and specified `context`.

None can be used as a wildcard.

size()
Return the number of *triples* in the current *store*.

update(**resources*)
Update the `*resources` to the *store* - persist.

2.4.2 The `surf.plugin.Plugin` Base Class

class `surf.plugin.`**Plugin**(**args*, ***kwargs*)
Bases: `object`

Super class for all SuRF plugins, provides basic instantiation and *logging*.

close()
Close the *plugin* and free any resources it may hold.

enable_logging(*enable=True*)
Enables or disable *logging* for the current *plugin*.

inference
Toggle *logical inference* on / off. The property has any effect only if such functionality is supported by the underlying data *store*.

is_enable_logging()
True if *logging* is enabled.

2.5 The `surf.query` Module

class `surf.query.`**Filter**
Bases: `unicode`

A **SPARQL** triple pattern filter

class `surf.query.`**Group**
Bases: `list`

A **SPARQL** triple pattern group

class `surf.query.`**NamedGroup**(*name=None*)
Bases: `surf.query.Group`

A **SPARQL** triple pattern named group

class `surf.query.`**OptionalGroup**
Bases: `surf.query.Group`

A **SPARQL** triple pattern optional group

class `surf.query.`**Query**(*type*, **vars*)
Bases: `object`

The *Query* object is used by SuRF to construct queries in a programatic manner. The class supports the major SPARQL query types: *select*, *ask*, *describe*, *construct*. Although it follows the SPARQL format the query can be translated to other Query formats such as PROLOG, for now though only SPARQL is supported.

Query objects should not be instatiated directly, instead use module-level `ask()`, `construct()`, `describe()`, `select()` functions.

Query methods can be chained.

distinct()
Add *DISTINCT* modifier.

filter(*filter*)
Add *FILTER* construct to query *WHERE* clause.

`filter` must be either *string/unicode* or `surf.query.Filter` object, if it is *None* then no filter is appended.

from_(**uris*)
Add graph URI(s) that will go in separate *FROM* clause.

Each argument can be either *string* or `surf.rdf.URIRef`.

from_named(**uris*)
Add graph URI(s) that will go in separate *FROM NAMED* clause.

Each argument can be either *string* or `surf.rdf.URIRef`.

limit(*limit*)
Add *LIMIT* modifier to query.

named_group(*name*, **statements*)
Add `GROUP ?name { ... }` construct to *WHERE* clause.

`name` is the variable name that will be bound to graph IRI.

`*statements` is one or more graph patterns.

Example:

```
>>> import surf
>>> from surf.query import a, select
>>> query = select("?s", "?src").named_group("?src", ("?s", a, surf.ns.FOAF['Person']))
>>> print unicode(query)
SELECT  ?s ?src  WHERE {  GRAPH ?src {  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
```

offset(*offset*)
Add *OFFSET* modifier to query.

optional_group(**statements*)
Add optional group graph pattern to *WHERE* clause.

optional_group() accepts multiple arguments, similarly to `where()`.

order_by(**vars*)
Add *ORDER_BY* modifier to query.

query_data
the query *data*, internal structure representing the contents of the *WHERE* clause

query_from
list of URIs that will go into query FROM clauses

query_from_named
list of URIs that will go into query FROM NAMED clauses

query_limit
the query *limit*, can be a number or None

query_modifier
the query *modifier* can be: *DISTINCT*, *REDUCED*, or *None*

query_offset
the query *offset*, can be a number or None

query_order_by
the query *order by* variables

query_type
the query *type* can be: *SELECT*, *ASK*, *DESCRIBE*or *CONSTRUCT*

query_vars
the query *variables* to return as the resultset

reduced()
Add *REDUCED* modifier.

where(**statements*)
Add graph pattern(s) to *WHERE* clause.

where() accepts multiple arguments. Each argument represents a a graph pattern and will be added to default group graph pattern. Each argument can be *tuple*, *list*, `surf.query.Query`, `surf.query.NamedGroup`, `surf.query.OptionalGroup`.

Example:

```
>>> query = select("?s").where(("?s", a, surf.ns.FOAF["person"]))
```

class surf.query.**Union**
Bases: `surf.query.Group`

A **SPARQL** union

surf.query.**ask**()
Construct and return `surf.query.Query` object of type **ASK**

surf.query.**construct**(**vars*)
Construct and return `surf.query.Query` object of type **CONSTRUCT**

surf.query.**describe**(**vars*)
Construct and return `surf.query.Query` object of type **DESCRIBE**

surf.query.**group**(**statements*)
Return group graph pattern.

Returned object can be used as argument in `Query.where()` method.

group()‘ accepts multiple arguments, similarly to `Query.where()`.

surf.query.**named_group**(*name*, **statements*)
Return named group graph pattern.

Returned object can be used as argument in `Query.where()` method.

`*statements` is one or more graph patterns.

Example:

```
>>> import surf
>>> from surf.query import a, select, named_group
>>> query = select("?s", "?src").where(named_group("?src", ("?s", a, surf.ns.FOAF['Person'])))
>>> print unicode(query)
SELECT  ?s ?src  WHERE {  GRAPH ?src {  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <ht
```

surf.query.**optional_group**(**statements*)
Return optional group graph pattern.

Returned object can be used as argument in `Query.where()` method.

optional_group() accepts multiple arguments, similarly to `Query.where()`.

surf.query.**select**(**vars*)
Construct and return `surf.query.Query` object of type **SELECT**

`*vars` are variables to be selected.

Example:

```
>>> query = select("?s", "?p", "?o")
```

surf.query.**union**(**statements*)
Return union graph pattern.

Returned object can be used as argument in `Query.where()` method.

union()‘ accepts multiple arguments, similarly to `Query.where()`.

2.6 The `surf.resource` Module

2.6.1 Contents

The `surf.resource.value` Module

class surf.resource.value.**ResourceValue**(*values_source*, *resource*, *attribute_name*)
Bases: `list`

the `surf.resource.value.ResourceValue` class is used by the `surf.resource.Resource` class to *lazy load* instances of resources.

Note: the class also emulates a list, while in addition providing support for *SuRF* queries, as defined in the `surf.query` module.

Note: instances of this class **must** not be created manually, instead they are automatically generated by *SuRF* as needed

---

context(*context*)
get the *context query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

count
L.count(value) -> integer – return number of occurrences of value

desc()
get the *desc query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

first
return the first *resource* or None otherwise.

full (*only_direct=False*)
get the *full query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

get_by (***kwargs*)
get the *get_by query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

get_first ()
return the first *resource* or None otherwise.

get_one ()
return only one *resource*. If there are more *resources* available the `surf.exc.NoResultFound` exception is raised

index
L.index(value, [start, [stop]]) -> integer – return first index of value. Raises ValueError if the value is not present.

limit (*value*)
get the *limit query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

offset (*value*)
get the *offset query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

one
return only one *resource*. If there are more *resources* available the `surf.exc.NoResultFound` exception is raised

order (*value=True*)
get the *order query* attribute. Syntactic sugar for `surf.resource.Resource.query_attribute()` method.

reverse
L.reverse() – reverse *IN PLACE*

set_dirty (*dirty*)
mark this *resource* as **dirty**. By doing so, *SuRF* will refresh it's content as soon as it's necessary

sort
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

to_rdf (*value*)
return an **RDF** representation of the *resource*

The `surf.resource.result_proxy` Module

Module for ResultProxy.

class `surf.resource.result_proxy.`**ResultProxy** (*params={}*, *store=None*, *instancemaker=None*)

Bases: `object`

Interface to `surf.store.Store.get_by()`.

ResultProxy collects filtering parameters. When iterated, it executes `surf.store.Store.get_by()` with collected parameters and yields results.

ResultProxy doesn't know how to convert data returned by `surf.store.Store.get_by()` into `surf.resource.Resource`, *URIRef* and *Literal* objects. It delegates this task to *instancemaker* function.

context (*context*)
Specify context/graph that resources should be loaded from.

desc ()
Set sorting order to descending.

filter (***kwargs*)
Add filter conditions.

Expects arguments in form:

```
ns_predicate = "(%s > 15)"
```

`ns_predicate` specifies which predicate will be used for filtering, a query variable will be bound to it. *%s* is a placeholder for this variable.

Filter expression (in example: "(%s > 15)") must follow SPARQL specification, on execution "%s" will be substituted with variable and the resulting string will be placed in query as-is. Because of string substitution percent signs need to be escaped. For example:

```
Person.all().filter(foaf_name = "(%s LIKE 'J%%')")
```

This Virtuoso-specific filter is intended to select persons with names starting with "J". In generated query it will look like this:

```
...
?s <http://xmlns.com/foaf/0.1/name> ?f1 .
FILTER (?f1 LIKE 'J%')
...
```

first ()
Return first resource or None if there aren't any.

full (*only_direct=False*)
Enable eager-loading of resource attributes.

If `full` is set to *True*, returned resources will have attributes already loaded.

Whether setting this will bring performance improvements depends on reader plugin implementation. For example, sparql_protocol plugin is capable of using SPARQL subqueries to fully load multiple resources in one request.

get_by (***kwargs*)
Add filter conditions.

Arguments are expected in form:

```
foaf_name = "John"
```

Multiple arguments are supported. An example that retrieves all persons named "John Smith":

```
FoafPerson = session.get_class(surf.ns.FOAF.Person)
for person in FoafPerson.get_by(foaf_name = "John", foaf_surname = "Smith"):
    print person.subject
```

instancemaker (*instancemaker_function*)
Specify the function for converting triples into instances.

`instancemaker_function` function can also be specified as argument to constructor when instantiating `ResultProxy`.

`instancemaker_function` will be executed whenever `ResultProxy` needs to return a resource. It has to accept two arguments: `params` and `instance_data`.

`params` will be a dictionary containing query parameters gathered by `ResultProxy`. Information from `params` can be used by `instancemaker_function`, for example, to decide what context should be set for created instances.

`instance_data` will be a dictionary containing keys *direct* and *inverse*. These keys map to dictionaries describing direct and inverse attributes respectively.

limit(*value*)
Set the limit for returned result count.

offset(*value*)
Set the limit for returned results.

one()
Return the only resource or raise if resource count != 1.

If the query matches no resources, this method will raise `surf.exc.NoResultFound` exception. If the query matches more than one resource, this method will raise `surf.exc.MultipleResultsFound` exception.

order(*value=True*)
Request results to be ordered.

If no arguments are specified, resources will be ordered by their subject URIs.

If `value` is set to an URIRef, corresponding attribute will be used for sorting. For example, sorting persons by surname:

```
FoafPerson = session.get_class(surf.ns.FOAF.Person)
for person in FoafPerson.all().order(surf.ns.FOAF.surname):
    print person.foaf_name.first, person.foaf_surname.first
```

Currently only one sorting key is supported.

2.7 The `surf.resource` base Module

class `surf.resource.`**Resource**(*subject=None*, *block_auto_load=False*, *context=None*, *namespace=None*)
Bases: `object`

The Resource class, represents the transparent proxy object that exposes sets of RDF triples under the form of <s,p,o> and <s',p,s> as an object in python.

One can create resource directly by instantiating this class, but it is advisable to use the session to do so, as the session will create subclasses of Resource based on the <s,rdf:type,'concept'> pattern.

Triples that constitute a resource can be accessed via Resource instance attributes. SuRF uses the following naming convention for attribute names: *nsprefix_predicate*. Attribute name examples: "rdfs_label", "foaf_name", "owl_Class".

Resource instance attributes can be set and get. If get, they will be structures of type `surf.resource.value.ResourceValue`. This class is subclass of *list* (to handle situations when there are several triples with the same subject and predicate but different objects) and have some some special features. Since *ResourceValue* is subtype of list, it can be iterated, sliced etc.

`surf.resource.value.ResourceValue.first()` will return first element of list or *None* if list is empty:

```
>>> resource.foaf_knows = [URIRef("http://p1"), URIRef("http://p2")]
>>> resource.foaf_knows.first
rdflib.URIRef('http://p1')
```

`surf.resource.value.ResourceValue.one()` will return first element of list or will raise if list is empty or has more than one element:

```
>>> resource.foaf_knows = [URIRef("http://p1"), URIRef("http://p2")]
>>> resource.foaf_knows.one
Traceback (most recent call last):
    ....
Exception: list has more elements than one
```

When setting resource attribute, it will accept about anything and translate it to *ResourceValue*. Attribute can be set to instance of *URIRef*:

```
>>> resource.foaf_knows = URIRef("http://p1")
>>> resource.foaf_knows
[rdflib.URIRef('http://p1')]
```

Attribute can be set to list or tuple:

```
>>> resource.foaf_knows = (URIRef("http://p1"), URIRef("http://p2"))
>>> resource.foaf_knows
[rdflib.Literal(u'http://p1', lang=rdflib.URIRef('http://p2'))]
```

Attribute can be set to string, integer, these will be converted into instances of *Literal*:

```
>>> resource.foaf_name = "John"
>>> resource.foaf_name
[rdflib.Literal(u'John')]
```

Attribute can be set to another SuRF resource. Values of different types can be mixed:

```
>>> resource.foaf_knows = (URIRef("http://p1"), another_resource)
>>> resource.foaf_knows
[rdflib.URIRef('http://p1'), <surf.session.FoafPerson object at 0xad049cc>]
```

Initialize a Resource, with the *subject* (a URI - either a string or a URIRef).

If `subject` is None than a unique subject will be generated using the `surf.util.uuid_subject()` function. If `namespace` is specified, generated subject will be in that namespace.

`block_auto_load` will prevent the resource from autoloading all rdf attributes associated with the subject of the resource.

classmethod `all`()
Retrieve all or limited number of *instances*.

`bind_namespaces`(**namespaces*)
Bind the *namespace* to the *resource*.

Useful for pretty serialization of the resource.

`bind_namespaces_to_graph`(*graph*)
Bind the 'resources' registered namespaces to the supplied *graph*.

classmethod `concept`(*subject*, *store=None*)
Return the Resources *concept* uri (type).

If parameter `store` is specified, concept will be retrieved from there. If resource was retrieved via session, it contains reference to store it was retrieved from and this reference will be used. Otherwise, *sessions default_store* will be used to retrieve the *concept*.

`context`
Context (graph) where triples constituting this resource reside in.

In case of SPARQL and SPARUL, "context" is the same thing as "graph".

Effects of having context set:

- When resource as whole or its individual attributes are loaded, triples will be only loaded from this context.
- When resource is saved, triples will be saved to this context.
- When existence of resource is checked (`is_present()`), only triples in this context will be considered.

context attribute would be usually set by *store* or *session* when instantiating resource, but it can also be set or changed on already instantiated resource. Here is an inefficient but workable example of how to move resource from one context to another:

```
Person = surf.ns.FOAF["Person"]
john_uri = "http://example/john"

old_graph_uri = URIRef("http://example/old_graph")
new_graph_uri = URIRef("http://example/new_graph")

instance = session.get_resource(john_uri, Person, old_graph_uri)
instance.context = new_graph_uri
instance.save()

# Now john is saved in the new graph but we still have to delete it
# from the old graph.

instance = session.get_resource(john_uri, Person, old_graph_uri)
instance.remove()
```

`dirty`
Reflects the *dirty* state of the resource.

classmethod `get_by`(***filters*)
Retrieve all instances that match specified filters and class.

Filters are specified as keyword arguments, argument names follow SuRF naming convention (they take form *namespace_name*).

Example:

```
>>> Person = session.get_class(surf.ns.FOAF['Person'])
>>> johns = Person.get_by(foaf_name = u"John")
```

classmethod `get_by_attribute`(*attributes*, *context=None*)
Retrieve all *instances* from the data store that have the specified *attributes* and are of *rdf:type* of the resource class

classmethod `get_dirty_instances`()
Return all the unsaved (dirty) *instances* of type *Resource*.

`graph`(*direct=True*)
Return an *rdflib ConjunctiveGraph* represenation of the current *resource*

is_present()
Return True if the *resource* is present in data *store*.

Resource is assumed to be present if there is at least one triple having `subject` of this resource as subject.

load()

Load all attributes from the data store:

- direct attributes (where the subject is the subject of the resource)
- indirect attributes (where the object is the subject of the resource)

Note: This method resets the *dirty* state of the object.

---

load_from_source(*data=None*, *file=None*, *location=None*, *format=None*)
Load the *resource* from a source (uri, file or string rdf data).

classmethod namespace()
Return the *namespace* of the currenlt Resources class type.

namespaces
The namespaces.

query_attribute(*attribute_name*)
Return ResultProxy for querying attribute values.

rdf_direct
Direct predicates (*outgoing* predicates).

rdf_inverse
Inverse predicates (*incoming* predicates).

remove(*inverse=False*)
Remove the *resource* from the data *store*.

classmethod rest_api(*resources_namespace*)
Return a `surf.rest.Rest` class responsible for exposing **REST** api functions for integration into REST aware web frameworks.

Note: The REST API was modeled according to the *pylons* model but it is generic enough to eb used in other frameworks.

---

save()
Save the *resource* to the data *store*.

serialize(*format='xml'*, *direct=False*)
Return a serialized version of the internal graph represenatation of the resource, the format is the same as expected by rdflib's graph serialize method

supported formats:

- **n3**
- **xml**
- **json** (internal serializer)
- **nt**
- **turtle**

set (*graph*)
Load the *resource* from a *graph*. The *graph* must be a *rdflib ConjunctiveGraph* or *Graph*

subject
The subject of the resource.

classmethod to_rdf (*value*)
Convert any value to it's appropriate *rdflib* construct.

update ()
Update the resource in the data *store*.

This method does not remove other triples related to it (the inverse triples of type <s',p,s>, where s is the *subject* of the *resource*)

2.8 The surf.rest Module

class surf.rest.**Rest** (*resources_namespace*, *concept_class*)
Bases: object

The Rest class handles the generation of REST like methods to perform CRUD operations on a surf.resource.Resource class

note: The REST api exposed is designed in accordance with the REST controller used in *pylons* applications, it adheres to the REST specification and offers extra features

the *resource* is the surf.resource.Resource class for which the REST interface is exposed, the *resources_namespace* represents the URI that instances will be using as subjects

create (*json_params*)
REST : POST /: Create a new item, creates a new instance of the current *Resource* type

delete (*id*)
REST : DELETE /id: Delete an existing item. removes the denoted instance from the underlying *store*

edit (*id*, *json_params*)
REST : GET /id;edit: updates an instances attributes with the supplied parameters

index (*offset=None*, *limit=None*)
REST : GET /: All items in the collection, returns all *instances* for the current *Resource*

new (*json_params*)
REST : GET /new: Form to create a new item. creates a new instance of the current *Resource* type

show (*id*)
REST : GET /id: Show a specific item. show / retrieve the specified resource

update (*id*, *json_params*)
REST : PUT /id: Update an existing item., update an instnaces attributes with the supplied parameters

2.9 The surf.serializer Module

surf.serializer.**to_json** (*graph*)
serializes a *rdflib Graph* or *ConjunctiveGraph* to **JSON** according to the specification of rdf-json for further details please see the following:

http://n2.talis.com/wiki/RDF_JSON_Specification

2.9.1 Serialization Example

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://example.org/about">
    <dc:creator>Anna Wilder</dc:creator>
    <dc:title xml:lang="en">Anna's Homepage</dc:title>
    <foaf:maker rdf:nodeID="person" />
  </rdf:Description>
  <rdf:Description rdf:nodeID="person">
    <foaf:homepage rdf:resource="http://example.org/about" />
    <foaf:made rdf:resource="http://example.org/about" />
    <foaf:name>Anna Wilder</foaf:name>
    <foaf:firstName>Anna</foaf:firstName>
    <foaf:surname>Wilder</foaf:surname>
    <foaf:depiction rdf:resource="http://example.org/pic.jpg" />
    <foaf:nick>wildling</foaf:nick>
    <foaf:nick>wilda</foaf:nick>
    <foaf:mbox_sha1sum>69e31bbcf58d432950127593e292a55975bc66fd</foaf:mbox_sha1sum>
  </rdf:Description>
</rdf:RDF>
```

is represented in **RDF-JSON** as

```
{
    "http://example.org/about" : {
        "http://purl.org/dc/elements/1.1/creator" : [ { "value" : "Anna Wilder", "type" : "literal" }
        "http://purl.org/dc/elements/1.1/title"   : [ { "value" : "Anna's Homepage", "type" : "litera
        "http://xmlns.com/foaf/0.1/maker"         : [ { "value" : "_:person", "type" : "bnode" } ]
    } ,

    "_:person" : {
        "http://xmlns.com/foaf/0.1/homepage"      : [ { "value" : "http://example.org/about", "type"
        "http://xmlns.com/foaf/0.1/made"          : [ { "value" : "http://example.org/about", "type"
        "http://xmlns.com/foaf/0.1/name"          : [ { "value" : "Anna Wilder", "type" : "literal" }
        "http://xmlns.com/foaf/0.1/firstName"     : [ { "value" : "Anna", "type" : "literal" } ] ,
        "http://xmlns.com/foaf/0.1/surname"       : [ { "value" : "Wilder", "type" : "literal" } ] ,
        "http://xmlns.com/foaf/0.1/depiction"     : [ { "value" : "http://example.org/pic.jpg", "type
        "http://xmlns.com/foaf/0.1/nick"          : [
                                                      { "type" : "literal", "value" : "wildling"} ,
                                                      { "type" : "literal", "value" : "wilda" }
                                                    ] ,
        "http://xmlns.com/foaf/0.1/mbox_sha1sum"  : [ {  "value" : "69e31bbcf58d432950127593e292a559
    }
}
```

for a more detailed description and the serialization algorithm please visit:

- http://n2.talis.com/wiki/RDF_JSON_Specification

2.10 The `surf.session` Module

class `surf.session.`**`Session`**(*default_store=None*, *mapping={}*, *auto_persist=False*, *auto_load=False*)

Bases: `object`

The *Session* will manage the rest of the components in **SuRF**, it also acts as the type factory for surf, the

resources will walk the graph in a lazy manner based on the session that they are bound to (the last created session).

Create a new *session* object that handles the creation of types and instances, also the session binds itself to the *Resource* objects to allow the Resources to access the data *store* and perform *lazy loading* of results.

Note: The *session* object *behaves* like a *dict* when it comes to managing the registered *stores*.

---

auto_load
Toggle *auto_load* (no need to explicitly call *load*, *resources* are loaded from the *store* automatically on creation) on or off. Accepts boolean values.

auto_persist
Toggle *auto_persistence* (no need to explicitly call *commit*, *resources* are persisted to the *store* each time a modification occurs) on or off. Accepts boolean values.

close()
Close the *session*.

Note: It is good practice to close the *session* when it's no longer needed. Remember: upon closing session all resources will lose the ability to reference the session thus the store and the mapping.

---

commit()
Commit all the changes, update all the *dirty resources*.

default_store
The *default store* of the session.

See *default_store_key* to see how the *default store* is selected.

default_store_key
The *default store key* of the session.

If it is set explicitly on *session* creation it is returned, else the first *store key* is returned. If no *stores* are in the session None is returned.

enable_logging
Toggle *logging* on or off. Accepts boolean values.

get_class(*uri*, *store=None*, **classes*)
See surf.session.Session.map_type(). The *uri* parameter can be any of the following:

- a *URIRef*
- a *Resource*
- **a *string* of the form**
  - a URI
  - a Resource class name eg: *SiocPost*
  - a namespace_symbol type string eg: *sioc_post*

get_resource(*subject*, *uri=None*, *store=None*, *graph=None*, *block_auto_load=False*, *context=None*, **classes*)
Same as *map_type* but *set* the resource from the *graph*.

keys()
The *keys* that are assigned to the managed *stores*.

load_resource(*uri*, *subject*, *store=None*, *data=None*, *file=None*, *location=None*, *format=None*, **classes*)
Create a *instance* of the *class* specified by *uri*.

Also set the internal properties according to the ones by the specified source.

map_instance(*concept*, *subject*, *store=None*, *classes=[]*, *block_auto_load=False*, *context=None*)
Create a *instance* of the *class* specified by *uri* and *classes* to be inherited, see *map_type* for more information.

map_type(*uri*, *store=None*, **classes*)
Create and return a *class* based on the *uri* given.

Also will add the *classes* to the inheritance list.

2.11 The surf.store Module

class surf.store.**Store**(*reader=None*, *writer=None*, **args*, ***kwargs*)
Bases: object

The *Store* class is comprised of a reader and a writer, getting access to an underlying triple store. Also store specific parameters must be handled by the class, the plugins act based on various settings.

The *Store* is also the *plugin* manager and provides convenience methods for working with plugins.

add_triple(*s=None*, *p=None*, *o=None*, *context=None*)
See surf.plugin.writer.RDFWriter.add_triple() method.

clear(*context=None*)
See surf.plugin.writer.RDFWriter.clear() method.

close()
Close the *store*.

Both the *reader* and the *writer* plugins are closed. See surf.plugin.writer.RDFWriter.close() and surf.plugin.reader.RDFReader.close() methods.

concept(*resource*)
surf.plugin.reader.RDFReader.concept() method.

enable_logging(*enable*)
Toggle *logging* on or off.

execute(*query*)
see surf.plugin.query_reader.RDFQueryReader.execute() method.

execute_sparql(*sparql_query*, *format='JSON'*)
see surf.plugin.query_reader.RDFQueryReader.execute_sparql() method.

get(*resource*, *attribute*, *direct*)
surf.plugin.reader.RDFReader.get() method.

index_triples(***kwargs*)
See surf.plugin.writer.RDFWriter.index_triples() method.

instances_by_attribute(*resource*, *attributes*, *direct*, *context*)
surf.plugin.reader.RDFReader.instances_by_attribute() method.

is_enable_logging()
True if *logging* is enabled, False otherwise.

is_present(*resource*)
surf.plugin.reader.RDFReader.is_present() method.

load(*resource*, *direct*)
surf.plugin.reader.RDFReader.load() method.

load_triples(*context=None*, ***kwargs*)
See surf.plugin.writer.RDFWriter.load_triples() method.

remove(**resources*, ***kwargs*)
See surf.plugin.writer.RDFWriter.remove() method.

remove_triple(*s=None*, *p=None*, *o=None*, *context=None*)
See surf.plugin.writer.RDFWriter.remove_triple() method.

save(**resources*)
See surf.plugin.writer.RDFWriter.save() method.

set_triple(*s=None*, *p=None*, *o=None*, *context=None*)
See surf.plugin.writer.RDFWriter.set_triple() method.

size()
See surf.plugin.writer.RDFWriter.size() method.

update(**resources*)
See surf.plugin.writer.RDFWriter.update() method.

2.12 The surf.util Module

surf.util.**attr2rdf**(*attrname*)
Convert an *attribute name* in the form:

```
# direct predicate
instance1.foaf_name
# inverse predicate
instance2.if_foaf_title_of
```

to

```
<!-- direct predicate -->
<http://xmlns.com/foaf/spec/#term_name>
<!-- inverse predicate -->
<http://xmlns.com/foaf/spec/#term_title>
```

The function returns two values, the *uri* representation and True if it's a direct predicate or False if its an inverse predicate.

surf.util.**de_camel_case**(*camel_case*, *delim=' '*, *method=2*)
Adds spaces to a camel case string. Failure to space out string returns the original string.

surf.util.**is_attr_direct**(*attrname*)
True if it's a direct *attribute*

```
>>> util.is_attr_direct('foaf_name')
True
>>> util.is_attr_direct('is_foaf_name_of')
False
```

surf.util.**is_uri**(*uri*)
True if the specified string is a URI reference False otherwise

surf.util.**json_to_rdflib**(*obj*)
Convert a json result entry to an rdfLib type.

surf.util.**namespace_split**(*uri*)
Same as *uri_split*, but instead of the base of the uri, returns the registered *namespace* for this uri

```
>>> print util.namespace_split('http://mynamespace/ns#some_property')
(rdflib.URIRef('http://mynamespace/ns#'), 'some_property')
```

surf.util.**pretty_rdf**(*uri*)
Returns a string of the given URI under the form *namespace:symbol*, if *namespace* is registered, else returns an empty string

surf.util.**rdf2attr**(*uri*, *direct*)
Inverse of *attr2rdf*, return the attribute name, given the URI and whether it is *direct* or not.

```
>>> print rdf2attr('http://xmlns.com/foaf/spec/#term_name',True)
foaf_name
>>> print rdf2attr('http://xmlns.com/foaf/spec/#term_title',False)
if_foaf_title_of
```

surf.util.**single**
Descriptor for easy access to attributes with single value.

surf.util.**uri_split**(*uri*)
Split the *uri* into base path and remainder, the base is everything that comes before the last #‘ or / including it

```
>>> print util.uri_split('http://mynamespace/ns#some_property')
('NS1', 'some_property')
```

surf.util.**uri_to_class**(*uri*)
returns a *class object* from the supplied *uri*, used *uri_to_class* to get a valid class name

```
>>> print util.uri_to_class('http://mynamespace/ns#some_class')
surf.util.Ns1some_class
```

surf.util.**uri_to_classname**(*uri*)
handy function to convert a *uri* to a Python valid *class name*

```
>>> # prints Ns1some_class, where Ns1 is the namespace (not registered, assigned automatically)
>>> print util.uri_to_classname('http://mynamespace/ns#some_class')
Ns1some_class
```

surf.util.**uuid_subject**(*namespace=None*)
the function generates a unique subject in the provided *namespace* based on the `uuid.uuid4()` method, If *namespace* is not specified than the default *SURF* namespace is used

```
>>>  print util.uuid_subject(ns.SIOC)
http://rdfs.org/sioc/ns#1b6ca1d5-41ed-4768-b86a-42185169faff
```

surf.util.**value_to_rdf**(*value*)
Convert the value to an *rdflib* compatible type if appropriate.

CHAPTER

THREE

ACKNOWLEDGEMENTS

A great deal of thanks go to:

- Peteris Caune
- Christoph Burgmer

for their contributions and ideas put into *SuRF*.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S